

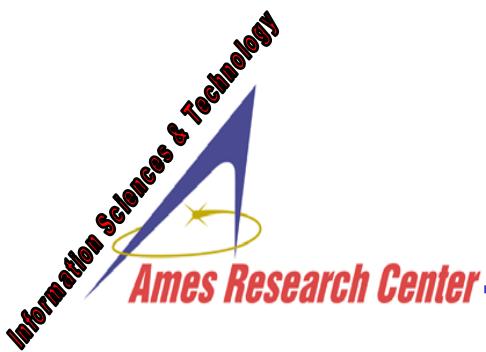
# Software Model Checking

Guillaume Brat, Dimitra Giannakopoulou, Klaus Havelund, Mike Lowry, Phil Oh, Corina Pasareanu,  
Charles Pecheur, John Penix, Willem Visser

and

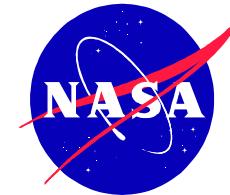
Matt Dwyer, John Hatcliff (Kansas State)  
Alex Groce, Flavio Llerda (CMU)

NASA Ames  
Automated Software Engineering Group



# Strategic Investments Research Program

## Technical Accomplishment



### High-Assurance Software Design

POC: Michael Lowry (Ames Research Center)

**Relevant Milestone:** Demonstrate scalable analytic verification technology on a major subsystem for Aerospace avionics. **(Project,Milestone,Date)**

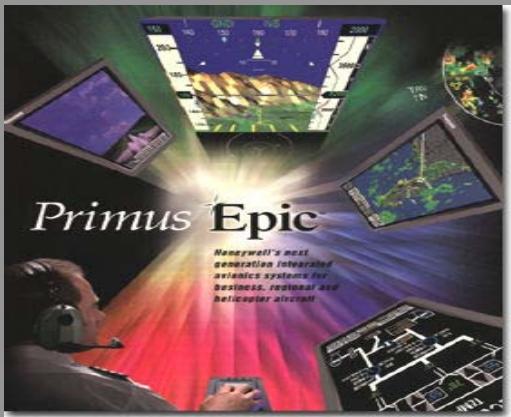
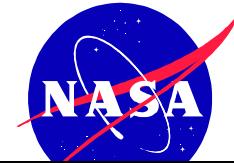
**Shown:** The application of model checking to the DEOS real-time embedded aerospace operating system from Honeywell to discover a subtle error not uncovered using the testing techniques required for FAA certification. This impact of this error during flight could have been starvation of critical real-time flight calculations. Indicate the scaling of model checking by showing the average factor of increase in lines of code (yellow) and state-space handled (white) by each technique developed and, in the middle, a graph indicating the impact of these techniques with respect to the time taken to analyze a 1000 lines of code.

**Accomplishment / Relation to Milestone and ETG:** Development of the Java PathFinder model checker, with accompanying set of synergistic verification technologies (including, abstractions, slicing, partial-order reduction, intelligent search and environment generation techniques) to enable the efficient analysis of object-oriented, concurrent programs such as those found in the next generation of avionics systems (e.g. the DEOS O/S for Integrated Modular Avionic systems). These model checking technologies have significantly reduced the effort required to analyze avionics software: currently we analyze 1000 lines of code per day compared to state of practice of 50 LOC/day in 1998.

**Future Plans:** Develop techniques to allow guarantees for correct behavior under certain assumptions that can be checked during actual execution using run-time program monitoring. Also, development of “learning” algorithms whereby the model checker’s search strategy can be adapted according to the structure of the program being analyzed.

2  
ETG: Provide increased confidence and lower the cost of development of next generation avionics software

# High-Assurance Software Design

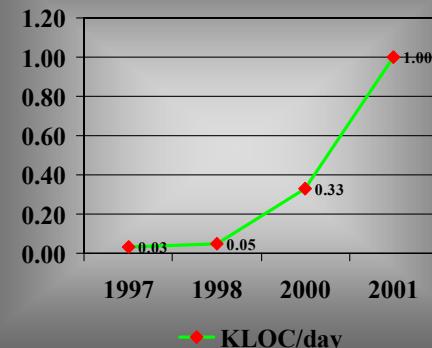


DEOS  
10000 lines to 1500

3x Slicing 30x  
Property preserving

Case 0:  
new();  
Case 1:  
Stop();  
Case 2:  
Remove();  
Case 3:  
Wait();

Combined techniques allows  
 $O(10^2)$  source line and  
 $O(10^6)$  state-space increase  
over state of practice

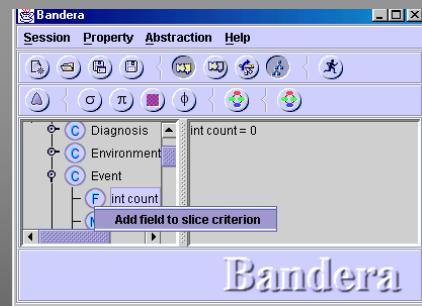


5x Abstraction 100x  
DEOS  
Infinite state to 1,000,000 states

Environment Generation

Semi-automated and requires domain knowledge

Bandera code-level debugging of error-path



Spurious error  
elimination during  
abstraction

2x Heuristic search  
10x Focused search for  
errors

JPF  
Model Checker

State compression  
2x 15x

Partial-order reduction  
2x 10x

Case 0:  
new();  
Case 2:  
Remove();

# Motivation



Mars Polar Lander

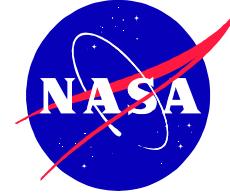


Ariane 501

Software Errors can be very costly

# Software

# Error-Detection



```

public boolean ConceptualObjectConstructor(int period) {
    itsPeriodIndex = period;
    itsCurrentPriority = Scheduler.priorityForPeriodIndex(its
    PeriodIndex);
    itsPeriodicEvent = StartOfPeriodEvent.eventForPer
    iodIndex(itsPeriodIndex);
    itsCurrentBudget = itsBudget;
    itsCreationStatus = threadStatusDormant;
    return true;
}

public void ConceptualObjectDestructor() {
    itsCreationStatus = threadStatusNotCreated;
}

public Budget budget() {
    return itsBudget;
}

public int currentPriority() {
    return itsCurrentPriority;
}

public void setCurrentPriority(int p) {
    //System.out.println("Thread.setCurrentPriority " + p);
    itsCurrentPriority = p;
}

public void startThread(int theCPUBudget) {
    //System.out.println("Thread.StartThread");
    itsCurrentPriority = Scheduler.priorityForPeriodIndex(
    itsPeriodIndex );
    itsBudget.setTotalBudgetInUsec(theCPUBudget);
    startThreadInternal();
    itsLastCompletion = itsPeriodicEvent.currentPeriod()-1;
    waitForNextTriggeringEvent(); // assumes critical!
    itsLastExecution = itsPeriodicEvent.currentPeriod();
    itsLastCompletion = itsPeriodicEvent.currentPeriod();
}

```

## Static Checking

### Type checking

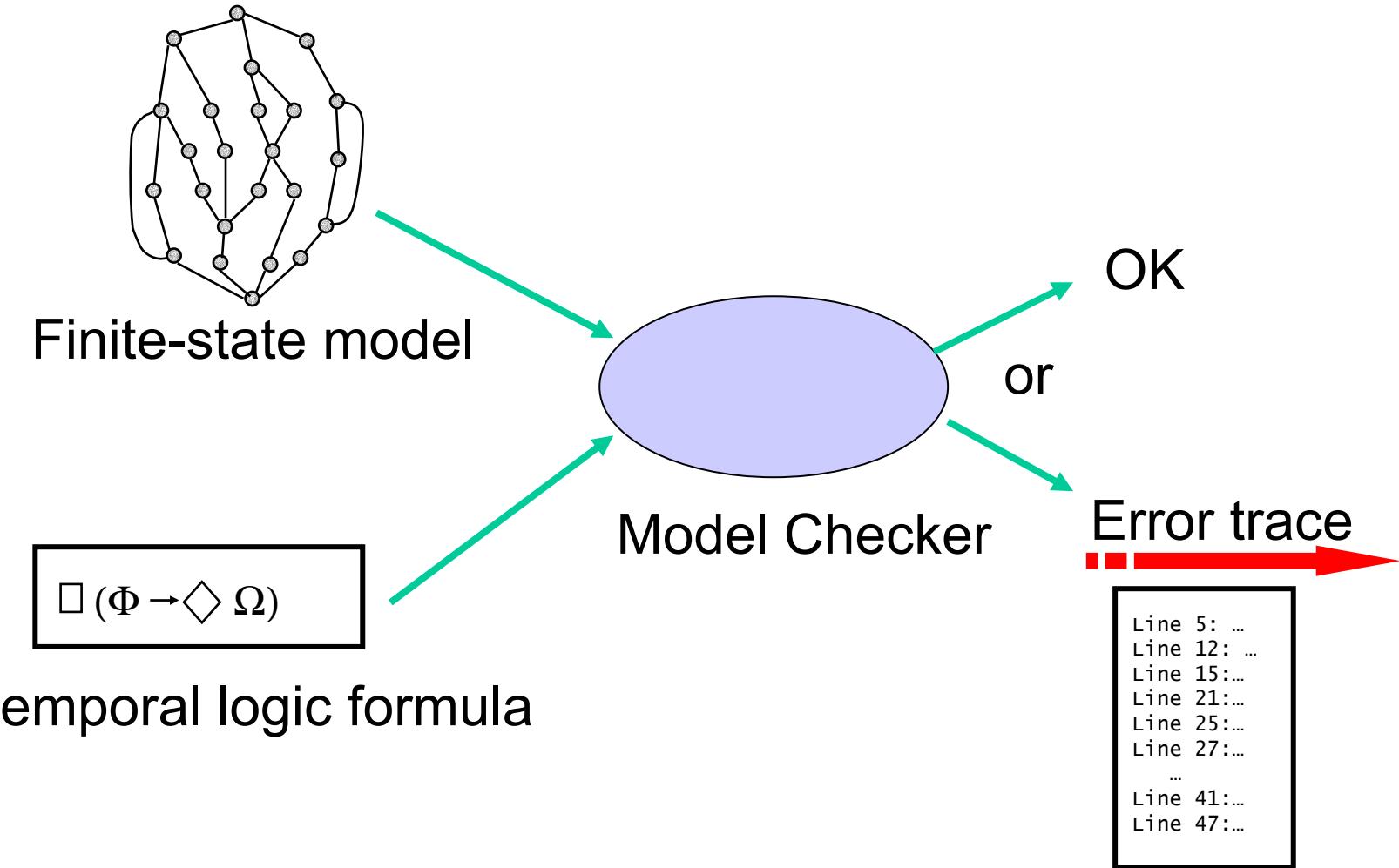
### Runtime-error Checking

## Dynamic Checking

### Testing

### Model Checking

# Model Checking



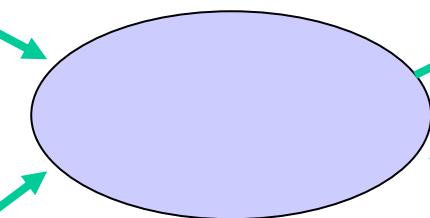
# The Dream

```
void add(Object o) {  
    buffer[head] = o;  
    head = (head+1)%size;  
}  
  
Object take() {  
    tail=(tail+1)%size;  
    return buffer[tail];  
}
```

Program

```
Property 1: ...  
Property 2: ...  
...
```

Requirement



Checker

OK

or

Error trace

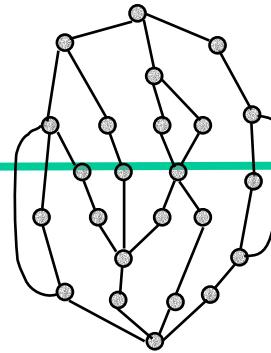


# Model Construction Problem

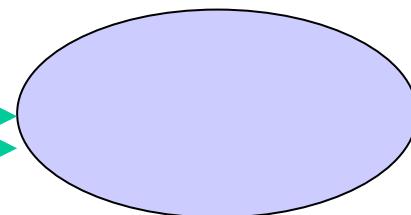


```
void add(Object o) {  
    buffer[head] = o;  
    head = (head+1)%size;  
}  
  
Object take() {  
    ...  
    tail=(tail+1)%size;  
    return buffer[tail];  
}
```

Program



Model Description



Model Checker

← Gap →

- **Semantic gap:**
  - Programming Languages

*methods, inheritance, dynamic creation, exceptions, etc.*

Model Description Languages

*automata*

# Research Agenda

- Goal
  - Demonstrate Scalable Analytic verification technology on a major aerospace subsystem
- Direction
  - Show model checking can be included in an iterative development cycle
  - Develop a model checker for Java
    - All the features of modern programming languages (objects, threads, exceptions etc.)
    - But none of the unnecessary complications (Pointers, direct memory access, etc.)
- Accomplishments
  - Java Pathfinder Model checker
  - Synergistic Verification technologies
  - Analysis of the DEOS Real-time Operating system

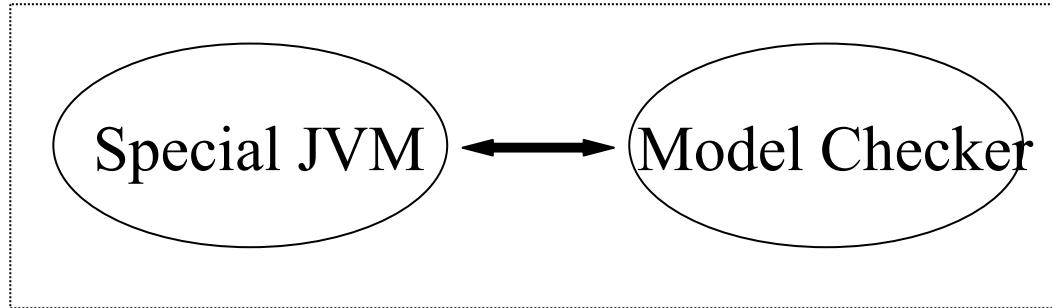
# Java PathFinder

```
void add(Object o) {  
    buffer[head] = o;  
    head = (head+1)%size;  
}  
  
Object take() {  
    ...  
    tail=(tail+1)%size;  
    return buffer[tail];  
}
```

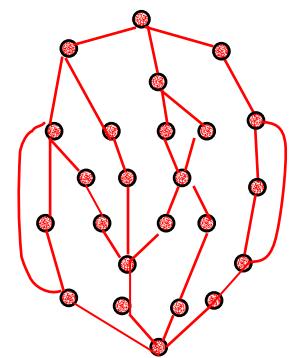
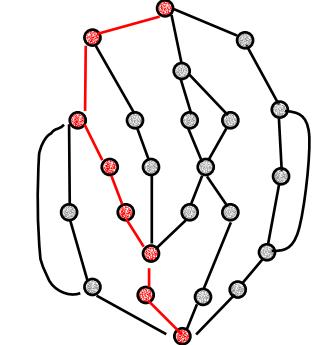
```
0:  iconst_0  
1:  istore_2  
2:  goto    #39  
5:  getstatic  
8:  aload_0  
9:  iload_2  
10: aaload
```

JVM

**Controllable Scheduler**



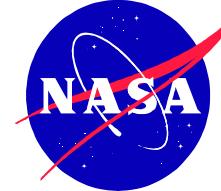
**Memory not Speed**  
**Modular Design**



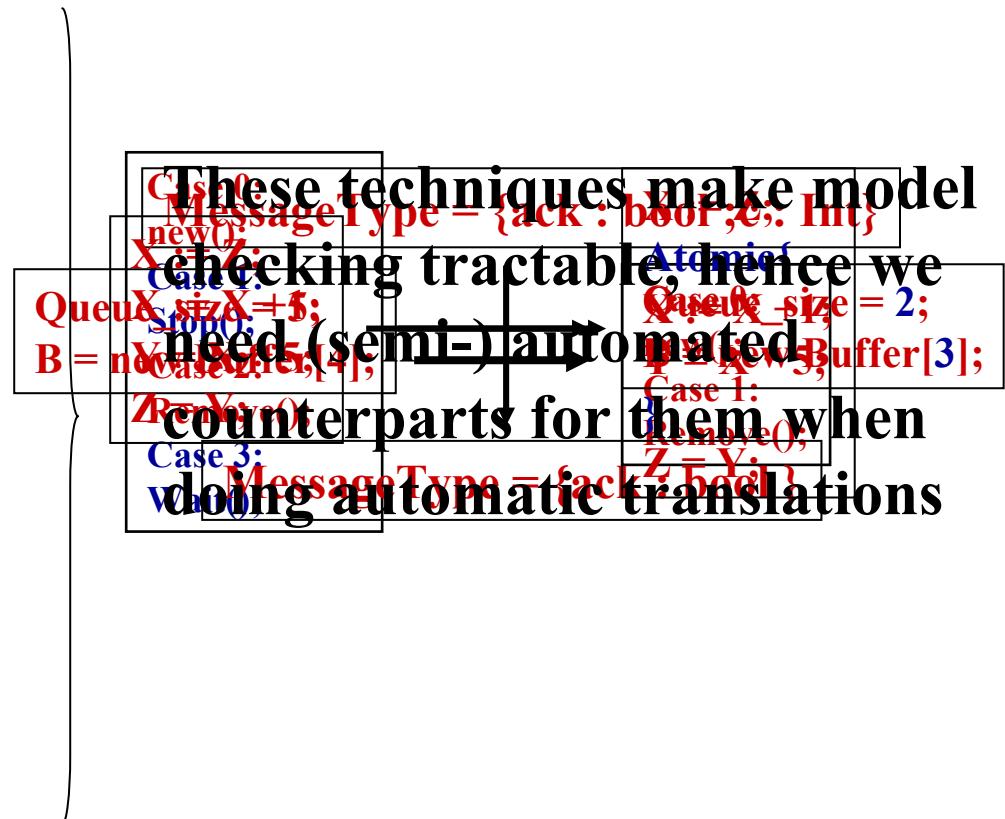
# Key Points

- Models can be infinite state
  - Depth-first state graph generation  
(Explicit-state model checking)
  - Errors are real
  - Verification can be problematic  
(Abstraction required)
- All of Java is handled except native code
- Nondeterministic Environments
  - JPF traps special nondeterministic methods
- Properties
  - User-defined assertions and deadlock
  - LTL properties (integrated with Bandera)
- Source level error analysis  
(with Bandera tool)

# Manual Code Simplification



- Remove *irrelevant* code
- Reduce sizes:  
e.g. Queues, arrays etc.
- Reduce variable ranges  
to singleton
- Group statements  
together in atomic blocks  
to reduce interleaving



# Enabling Technologies

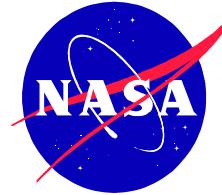
- Remove *irrelevant* code
- Reduce sizes:  
e.g. Queues, arrays etc.
- Reduce variable ranges  
to singleton
- Group statements  
together in atomic blocks  
to reduce interleaving

- Property Preserving **Slicing**
- **Abstraction**
  - Under-approximations
  - Over-approximations

- **Partial-order Reductions**
- State **Compression**
- **Heuristic Search**

# Technology Overview

---



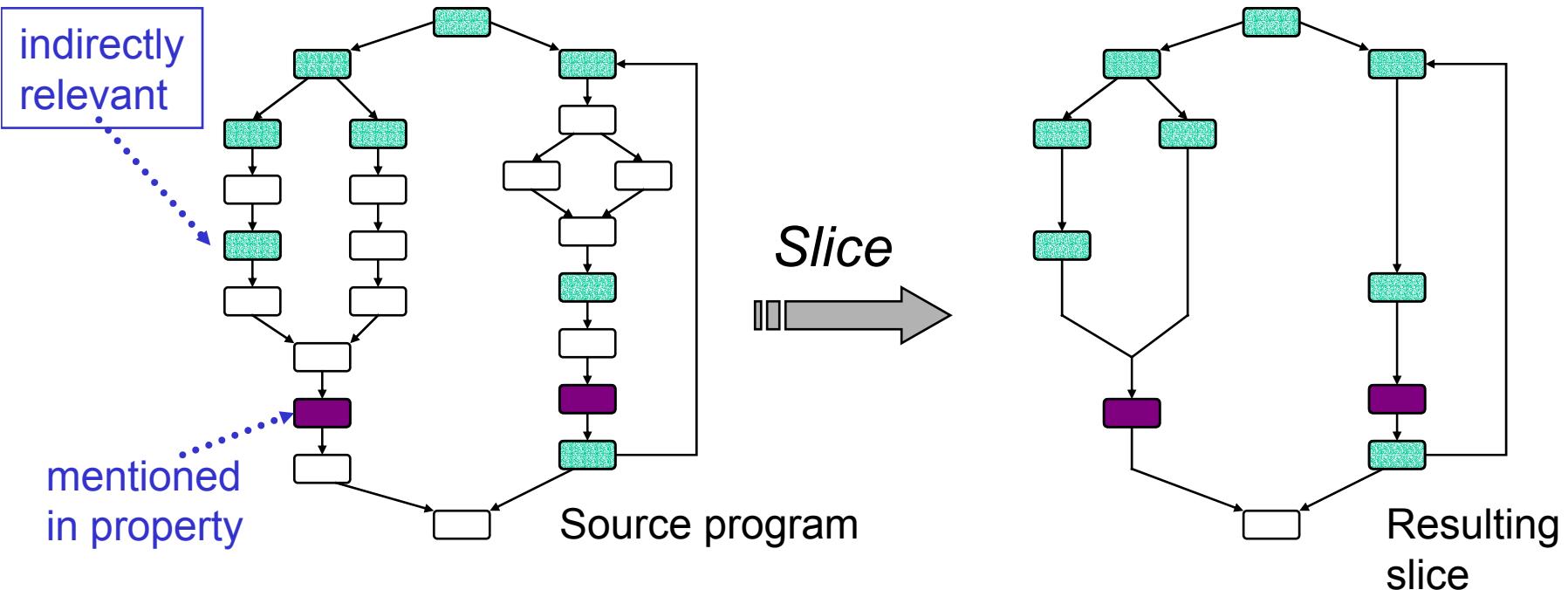
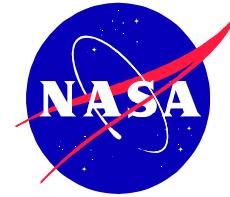
- Slicing
- Abstraction
- State Compression
- Partial-order Reduction
- Heuristic Search

# Technology Overview

---

- Slicing
- Abstraction
- State Compression
- Partial-order Reduction
- Heuristic Search

# Property-directed Slicing



- **slicing criterion** generated automatically from observables mentioned in the property
- backwards slicing automatically finds all components that might influence the observables. <sup>16</sup>

# Property-directed Slicing

```


/**
 * @observable EXP Full: (head == tail)
 */

class BoundedBuffer {
    Object[] buffer;
    int bound;
    int head, tail;

    public synchronized void add(Object o) {
        while (tail == head)
            try { wait(); } catch (InterruptedException ex) {}

        buffer[head] = o;
        head = (head+1) % bound;
        notifyAll();
    }
}


```

**Slicing Criterion**  
*All statements that assign to head, tail.*

Included in  
slicing  
criterion

removed by  
slicing

indirectly  
relevant

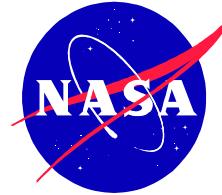
# Technology Overview

---

- Slicing
- Abstraction
  - Under-Approximations
  - Over-Approximations
    - Predicate Abstraction
    - Data Type Abstractions
- State Compression
- Partial-order Reduction
- Heuristic Search

# Technology Overview

---



- Slicing
- Abstraction
  - Under-Approximations
  - Over-Approximations
    - Predicate Abstraction
    - Data Type Abstractions
- State Compression
- Partial-order Reduction
- Heuristic Search

# Abstraction

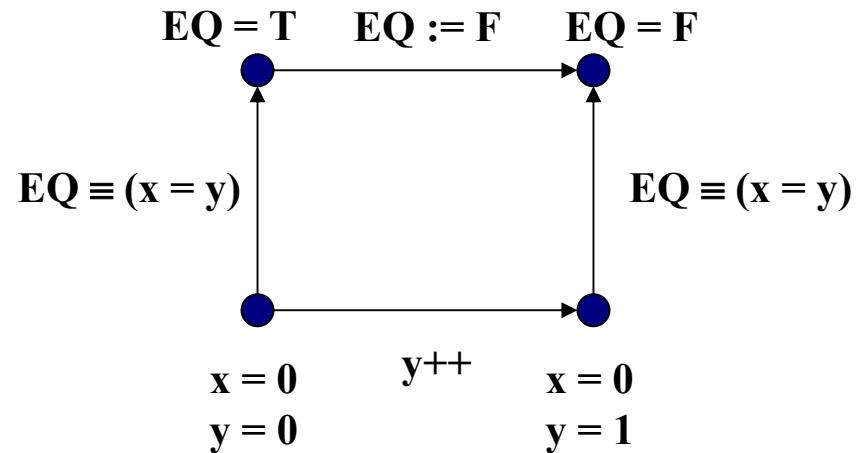
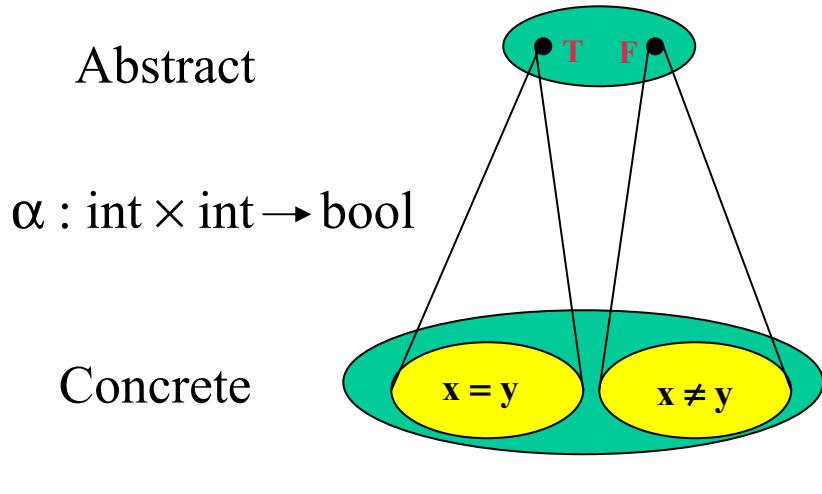
## Under-approximations

- Remove behaviors
- Preserves errors
  - “Exists” paths
- Transform code
  - Size changes
  - Manual
- Filtered Environments
  - Don’t allow all environment actions
  - Semi-automated

## Over-approximations

- Add behaviors
- Preserves correctness
  - “For all” paths
- Type-based abstractions
  - Semi-automated
- Predicate Abstraction
  - Semi-automated

# Predicate Abstraction



- Mapping of a concrete system to an abstract system, whose states correspond to truth values of a set of predicate

# Calculating Abstraction

Predicate:  $B \equiv (x = y)$

Concrete Statement

$y := y + 1$

Step 2a: Use Decision Procedures

$$\begin{array}{ll} x = y \rightarrow x = y + 1 & x = y \rightarrow x \neq y + 1 \\ x \neq y \rightarrow x = y + 1 & x \neq y \rightarrow x \neq y + 1 \end{array}$$

Abstract Statement

Step 1: Calculate pre-images

$$\begin{array}{lll} \{x = y + 1\} & y := y + 1 & \{x = y\} \\ \{x \neq y + 1\} & y := y + 1 & \{x \neq y\} \end{array}$$

Step 2: Rewrite in terms of predicates

$$\begin{array}{lll} \{x = y + 1\} & y := y + 1 & \{B\} \\ \{B\} & y := y + 1 & \{\sim B\} \end{array}$$

Step 3: Abstract Code

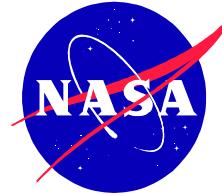
IF  $B$  THEN  $B := \text{false}$   
ELSE  $B := \text{true} \mid \text{false}$



JPF

# Predicate Abstraction

---



- Annotations used to indicate abstractions
  - `Abstract.remove(x);`  
`Abstract.remove(y);`  
`Abstract.addBoolean("EQ", x==y);`
- Tool generates abstract Java program
  - Using Stanford Validity Checker (SVC)
  - JVM is extended with nondeterminism to handle over approximation
- Abstractions can be local to a class or global across multiple classes
  - `Abstract.addBoolean("EQ", A.x==B.y);`
  - Dynamic predicate abstraction - works across instances

# Data Type Abstraction



Collapses data domains via abstract interpretation:

## Code

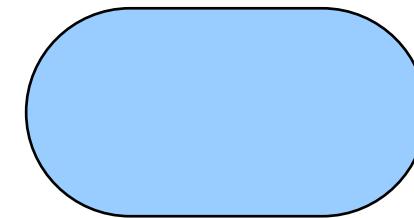
```
int x = 0;
if (x == 0)
    x = x + 1;
```



```
Signed x = ZERO;
if (Signed.eq(x, ZERO))
    x = Signed.add(x, POS);
```

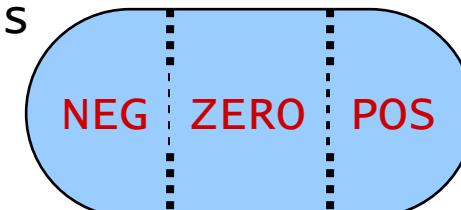
## Data domains

int



$(n < 0)$  : NEG  
 $(n == 0)$  : ZERO  
 $(n > 0)$  : POS

signs



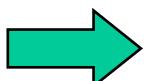
# Abstract Interpretation

```
abstraction Signs abstracts int
|-----|
TOKENS = { neg, zero, pos };
```

abstraction mapping:

n < 0	->	{neg};
n == 0	->	{zero};
n > 0	->	{pos};

+abs	zero	pos	neg
zero	zero	pos	neg
pos	pos	pos	{zero, pos, neg}
neg	neg	{zero, pos, neg}	neg



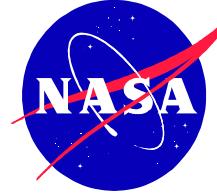
```
public class Signs {
    public static final int NEG = 0;
    public static final int ZERO = 1;
    public static final int POS = 2;

    public static int abs(int n) {
        if (n < 0) return NEG;
        if (n == 0) return ZERO;
        if (n > 0) return POS;
    }

    public static int add(int a, int b) {
        int r;
        Verify.beginAtomic();
        if (a==NEG && b==NEG) r=NEG;
        if (a==NEG && b==ZERO) r=NEG;
        if (a==ZERO && b==NEG) r=NEG;
        if (a==ZERO && b==ZERO) r=ZERO;
        if (a==ZERO && b==POS) r=POS;
        if (a==POS && b==ZERO) r=POS;
        if (a==POS && b==POS) r=POS;
        else r=Verify.choose(2);
        Verify.endAtomic();
        return r; }
```

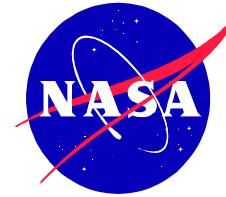
# Technology Overview

---



- Slicing
- Abstraction
  - Under-Approximations
  - Over-Approximations
    - Infeasible/Spurious Errors
- State Compression
- Partial-order Reduction
- Heuristic Search

# Example of Infeasible Counter-example



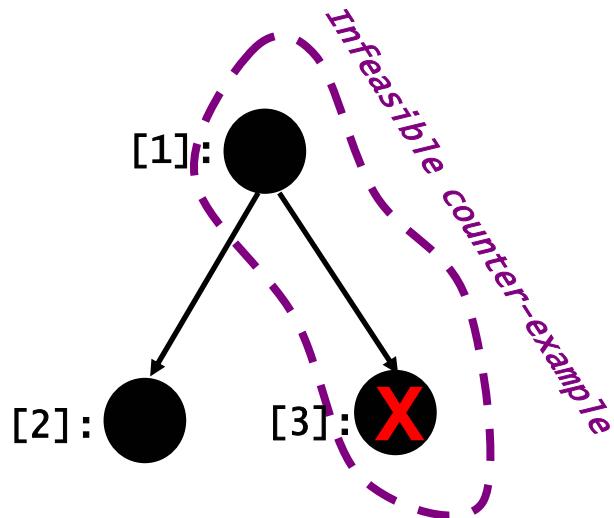
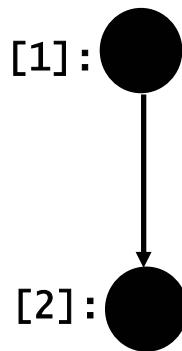
```
[1] if (-2 + 3 > 0)
then
[2]     assert(true);
else
[3]     assert(false);
```

**Signs:**

- n < 0 -> neg
- 0 -> zero
- n > 0 -> pos

{NEG, ZERO, POS}

```
[1] if(Signs.gt(signs.add(NEG, POS), ZERO))
then
[2]     assert(true);
else
[3]     assert(false);
```

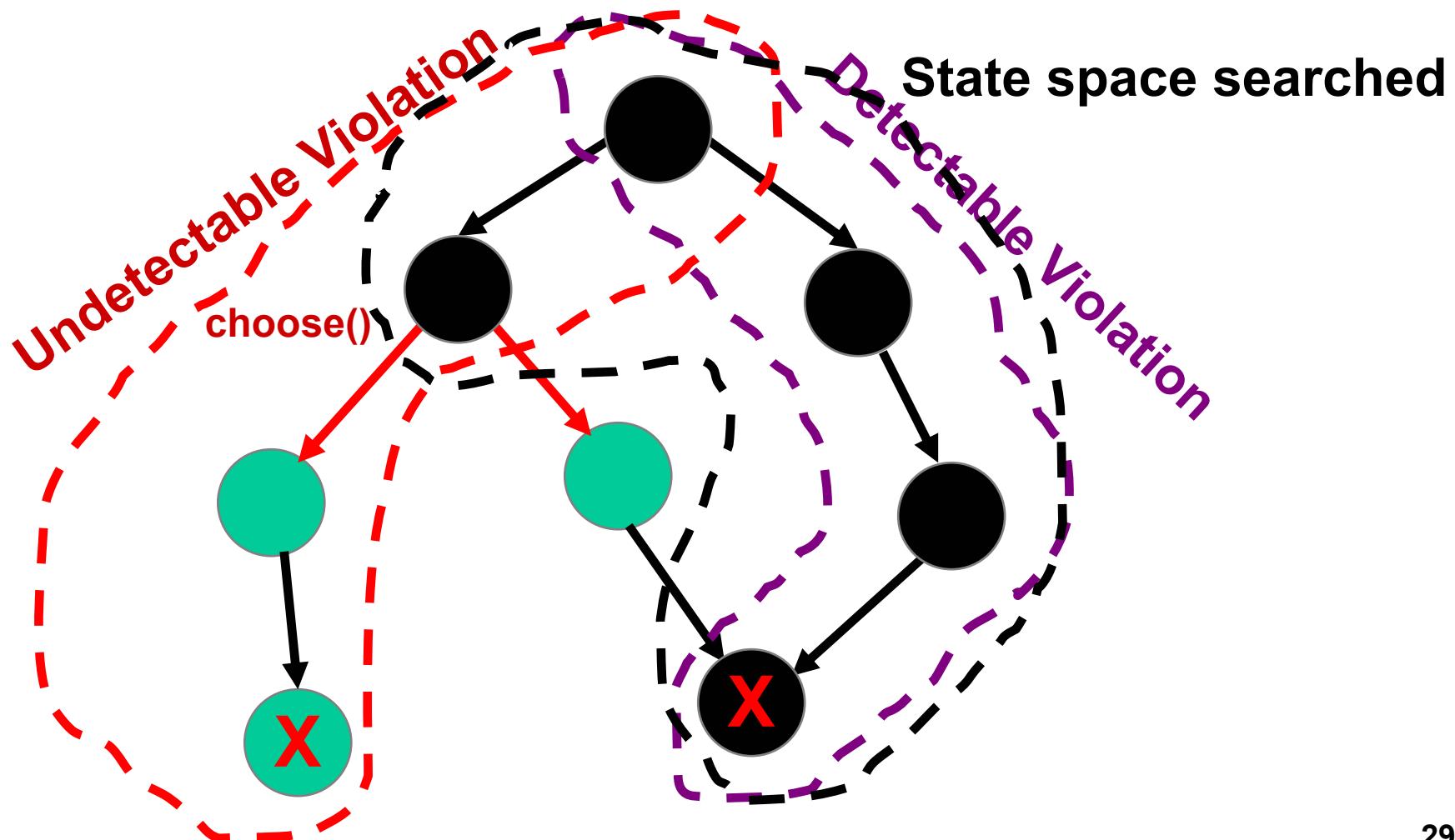


# Choose-free state space search



- Theorem [Saidi:SAS'00]  
Every path in the abstracted program where all assignments are deterministic is a path in the concrete program.
- Bias the model checker
  - to look only at paths that do not include instructions that introduce non-determinism
- JPF model checker modified
  - to detect non-deterministic choice and backtrack from those points

# Choice-bounded Search



# Technology Overview

---

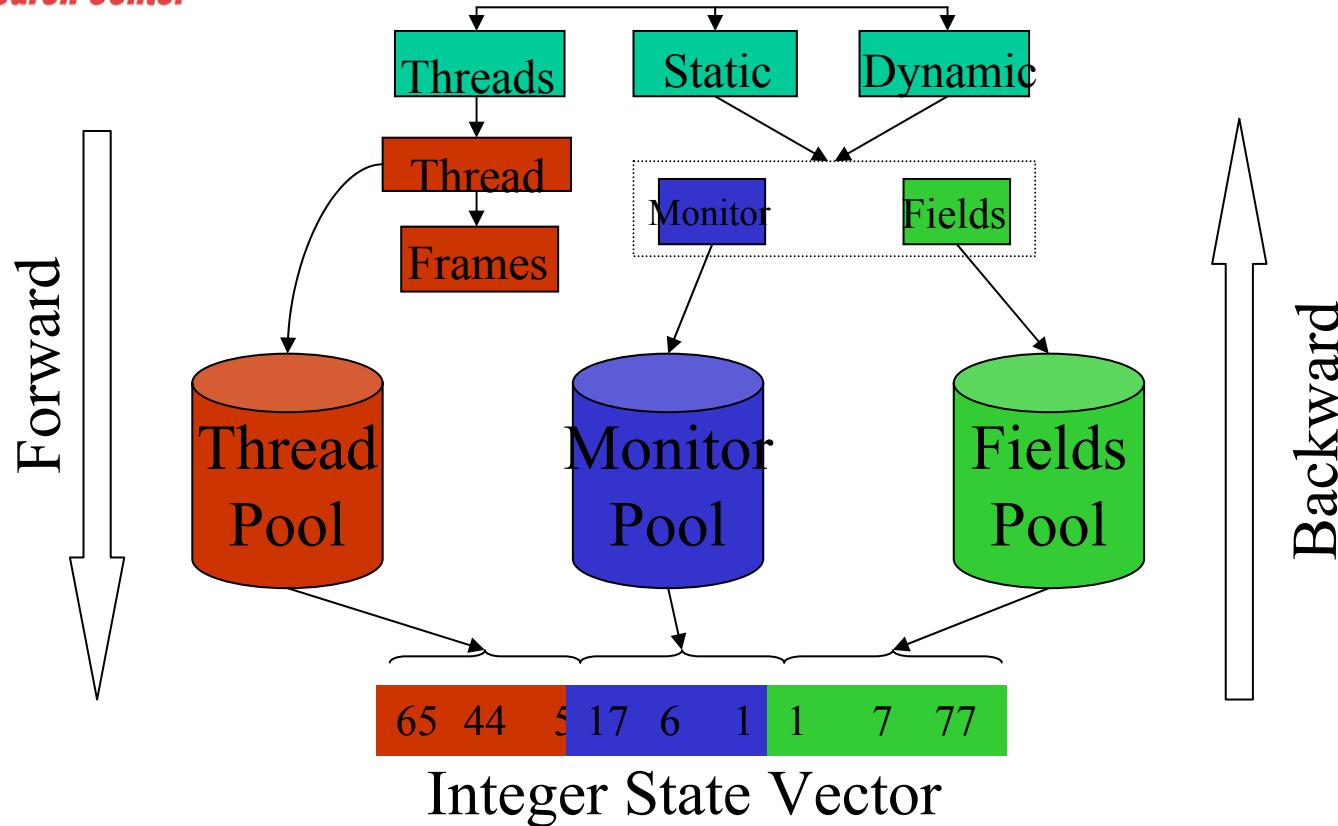
- Slicing
- Abstraction
  - Under-Approximations
  - Over-Approximations
    - Predicate Abstraction
    - Data Type Abstractions
- State Compression
- Partial-order Reduction
- Heuristic Search

# Technology Overview

---

- Slicing
- Abstraction
  - Under-Approximations
  - Over-Approximations
    - Predicate Abstraction
    - Data Type Abstractions
- State Compression
- Partial-order Reduction
- Heuristic Search

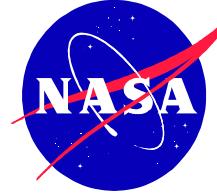
# State Compression



Allows **2x** source lines and **15x** state-space increase

# Technology Overview

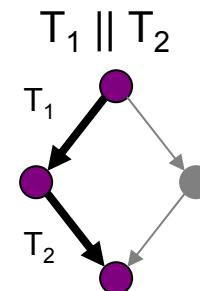
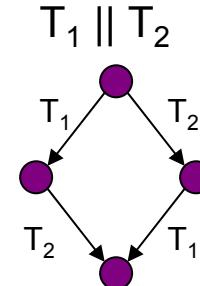
---



- Slicing
- Abstraction
  - Under-Approximations
  - Over-Approximations
    - Predicate Abstraction
    - Data Type Abstractions
- State Compression
- Partial-order Reduction
- Heuristic Search

# Reducing Interleavings

- Unnecessary Interleavings cause State-explosion
  - Interleaving independent statements
- Partial-order Reduction Eliminates unnecessary interleavings
  - Only interleave dependent statements during model checking
- Require Static analysis phase before model checking to determine statements that are globally independent
  - Advanced alias-analysis



# Partial-order Reduction

```

class S1 {int x;} class S2 {int y;}

public class Example {
    public static void main (String[] args) {
        FirstTask t1 = new FirstTask();
        SecondTask t2 = new SecondTask();
        t1.start(); t2.start();
    }
}

class FirstTask extends Thread {
    public void run() {
        int x; S1 s1;
        x = 1;
        s1 = new S1();
        x = 3;
    }
}

class SecondTask extends Thread {
    public void run() {
        int x; S2 s2;
        x = 2;
        s2 = new S2();
        x = 3;
    }
}

```

- **43** states with no reduction
- **18** states With partial-order reduction
  - all statements are globally independent (safe)

# Partial-order Reduction

```

class S1 {int x;} class S2 {int y;}
public class Example {
    public static int x = 10;
    public static void main (String[] args) {
        FirstTask t1 = new FirstTask();
        SecondTask t2 = new SecondTask();
        t1.start(); t2.start();
    }
}

class FirstTask extends Thread {
    public void run() {
        int x; S1 s1;
        Example.x = 1; ← Not Safe → Example.x = 2;
        s1 = new S1();
        x = 3;
    }
}

class SecondTask extends Thread {
    public void run() {
        int x; S2 s2;
        Example.x = 2;
        s2 = new S2();
        x = 3;
    }
}

```

- 43 states with no reduction
- 27 states With partial-order reduction
  - 2 statements are not globally independent

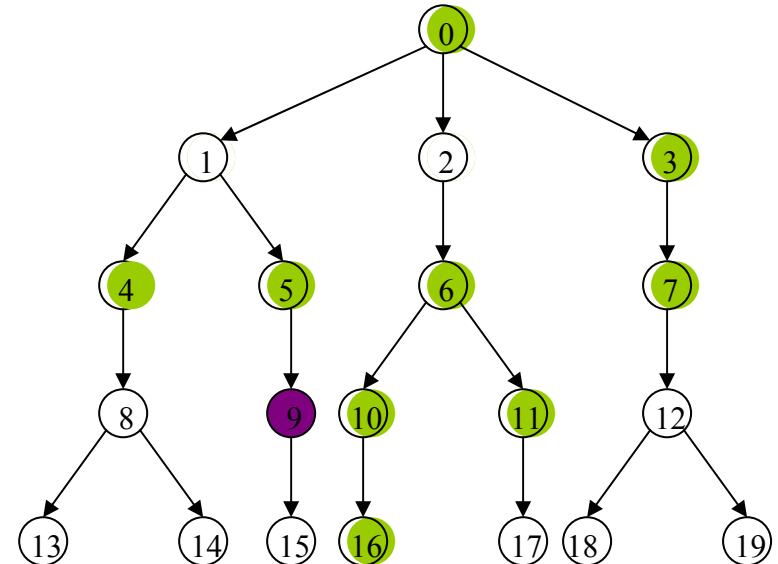
# Technology Overview

---

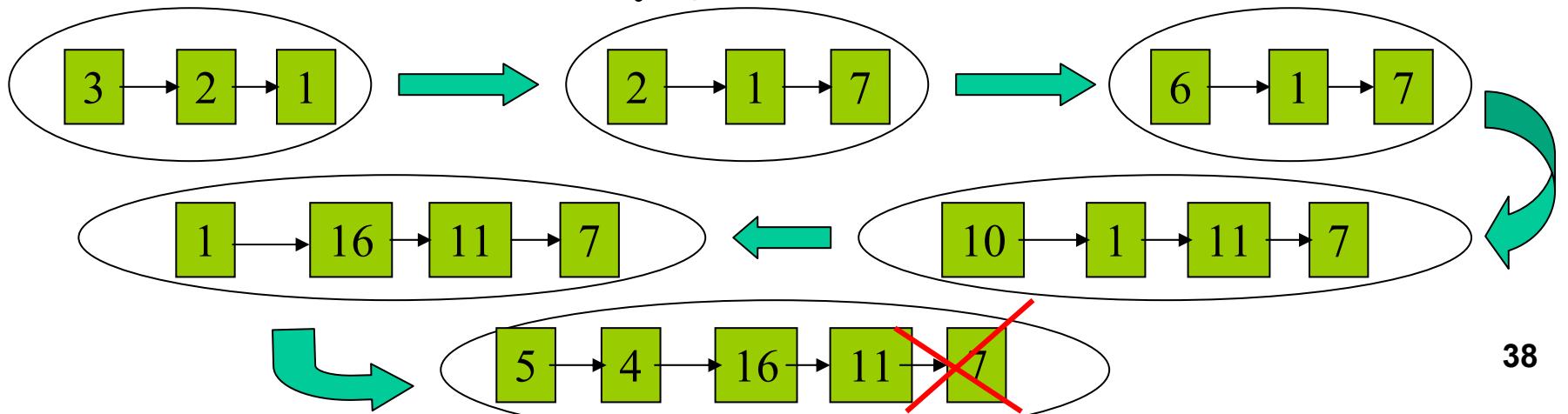
- Slicing
- Abstraction
- State Compression
- Partial-order Reduction
- Heuristic Search

# Heuristic Search

- Breadth-first (BFS) like state-generation
- Priority queue according to fitness function
- Queue limit parameter



**Priority Queue with limit 4**

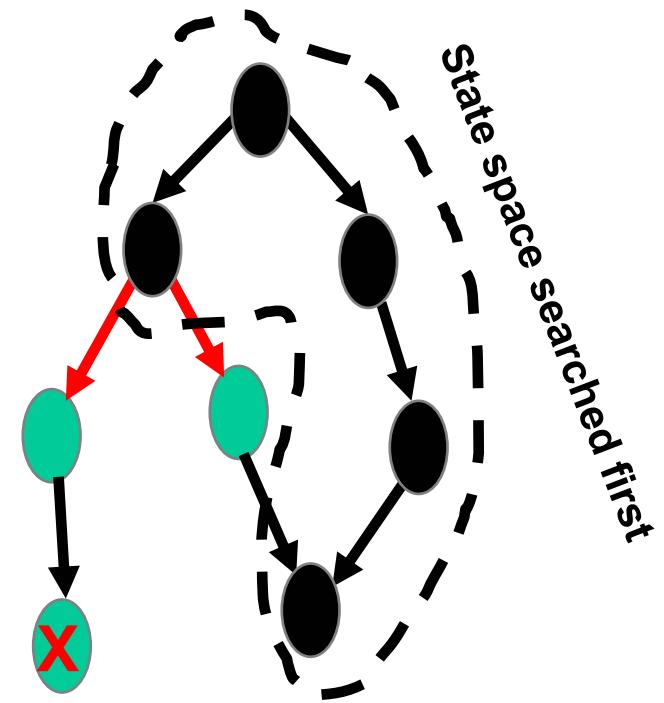


# Heuristic Search

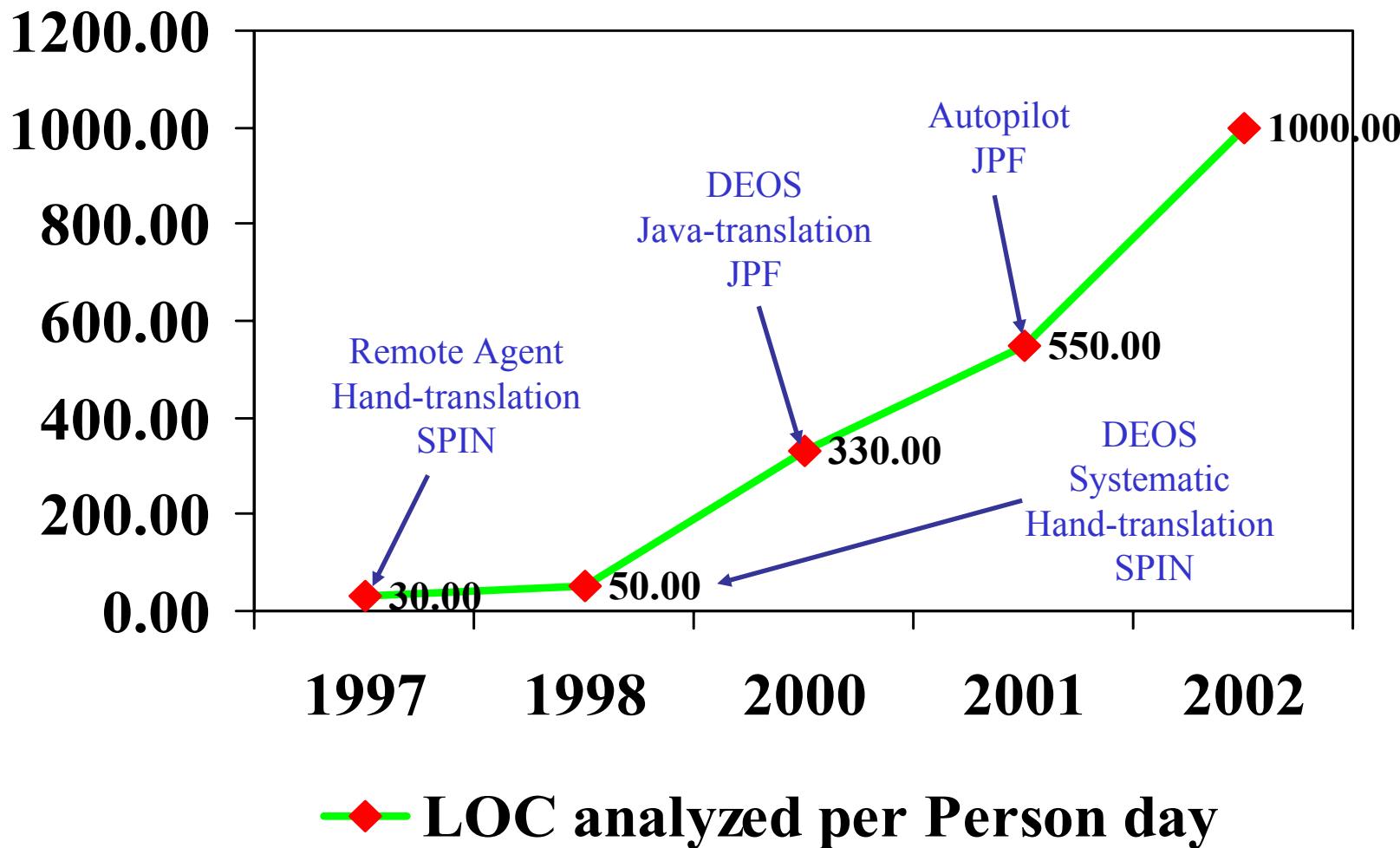
- Best-First, Beam and A\* Search
- Heuristics based on property
  - deadlock
    - Maximize number of blocked threads
  - Assertions
    - Minimize distance to assertion
- Heuristics on structure of Program
  - Interleaving heuristic
    - Maximize different thread scheduling
  - Branch Exploration
    - Maximize the coverage of new branches
  - Choose-free heuristic
    - Minimize non-deterministic choice
- User-defined heuristics
  - Full access to JVM's state via API
- Combine heuristics

# Choose-free Heuristic

- Infeasible error elimination during abstraction
- Heuristic function returns **best value** for states with **least number of non-deterministic choices enabled**
- If no “deterministic” error exists it also searches rest of the state space



# Scaling Program Model Checking Error-Detection



# JPF Released

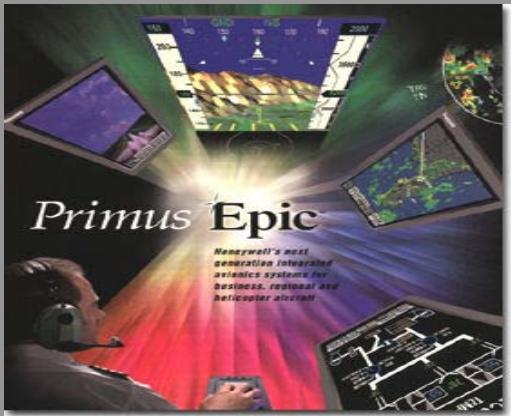
JPF released to  
collaborators and beta testers  
in February 2001

40 worldwide downloads

# Future Work

- Combined Property and Heuristic specification languages
  - “DFS until full(queue) then show no-deadlock using branch-exploration”
  - Allow model checker to “learn” how to search the state-space
- Combine Coverage, Model Checking and Runtime analysis to give bounded correctness guarantees
  - Check a system under certain environment assumptions, if property holds, then use runtime analysis to check assumptions during execution
- C/C++ Version under development

# High-Assurance Software Design

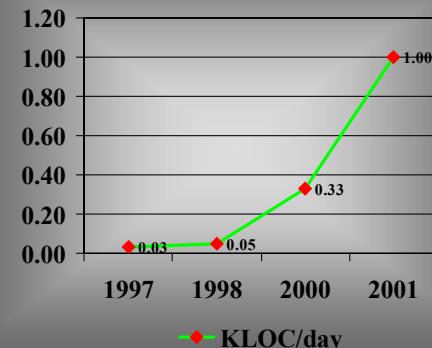


DEOS  
10000 lines to 1500

3x Slicing 30x  
Property preserving

Case 0:  
new();  
Case 1:  
Stop();  
Case 2:  
Remove();  
Case 3:  
Wait();

Combined techniques allows  
 $O(10^2)$  source line and  
 $O(10^6)$  state-space increase  
over state of practice

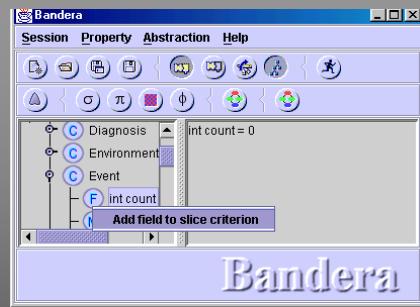


5x Abstraction DEOS  
Infinite state to 1,000,000 states

Environment Generation

Semi-automated and requires domain knowledge

Bandera code-level debugging of error-path



Spurious error  
elimination during  
abstraction

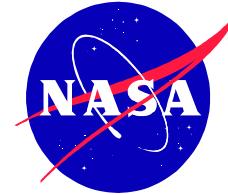
2x Heuristic search  
10x Focused search for  
errors

JPF  
Model Checker

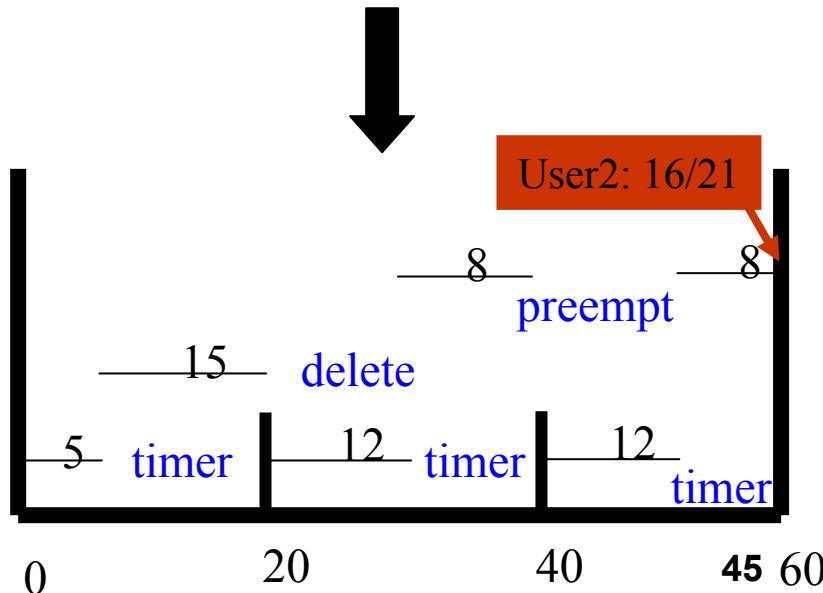
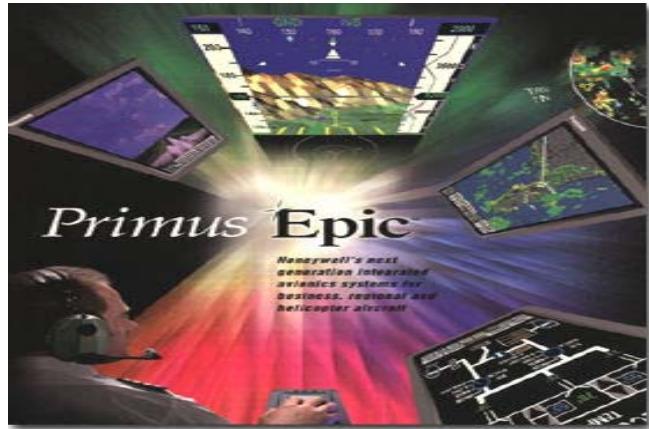
State compression  
2x 15x  
Partial-order reduction  
2x 10x

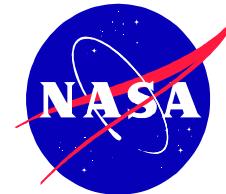
Case 0:  
new();  
Case 2:  
Remove();

# DEOS

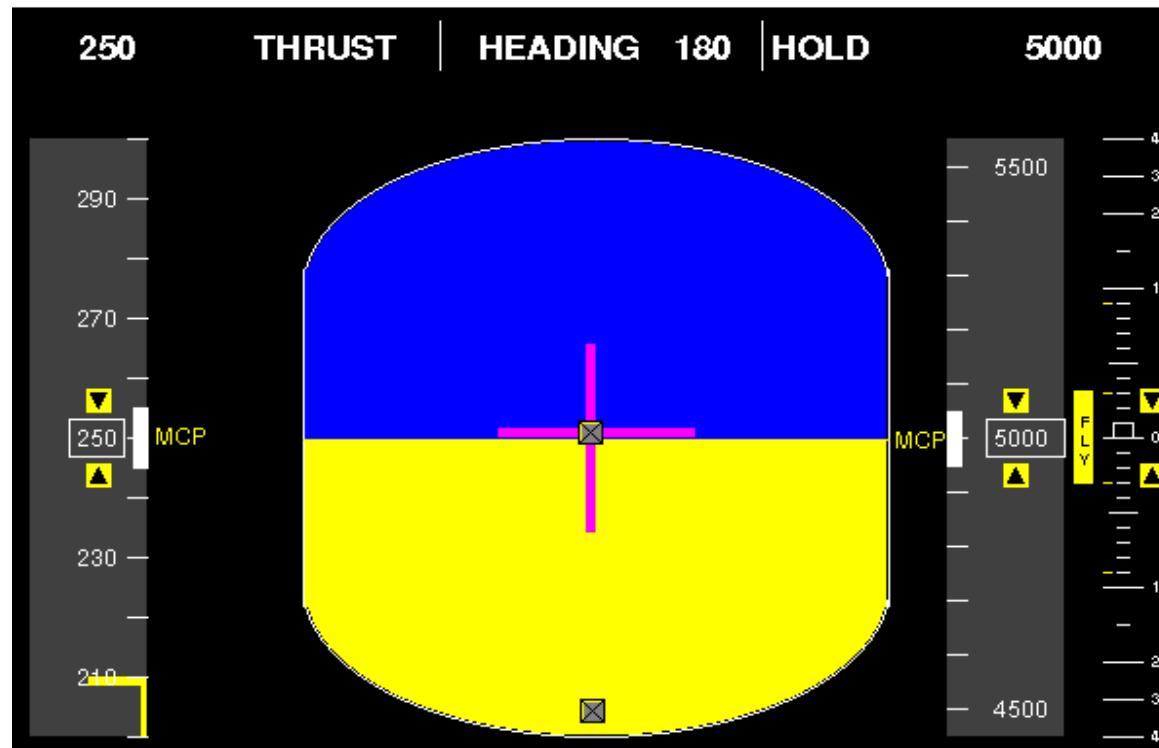


- Real-time O/S from Honeywell
- Subtle error
- 1500 lines of Java
  - C++ originally
- Dependency Analysis
- Apply Type Abstraction
- Spurious errors exist
- Use choose-free heuristic





# Autopilot Tutor



# Model Checking the Autopilot

